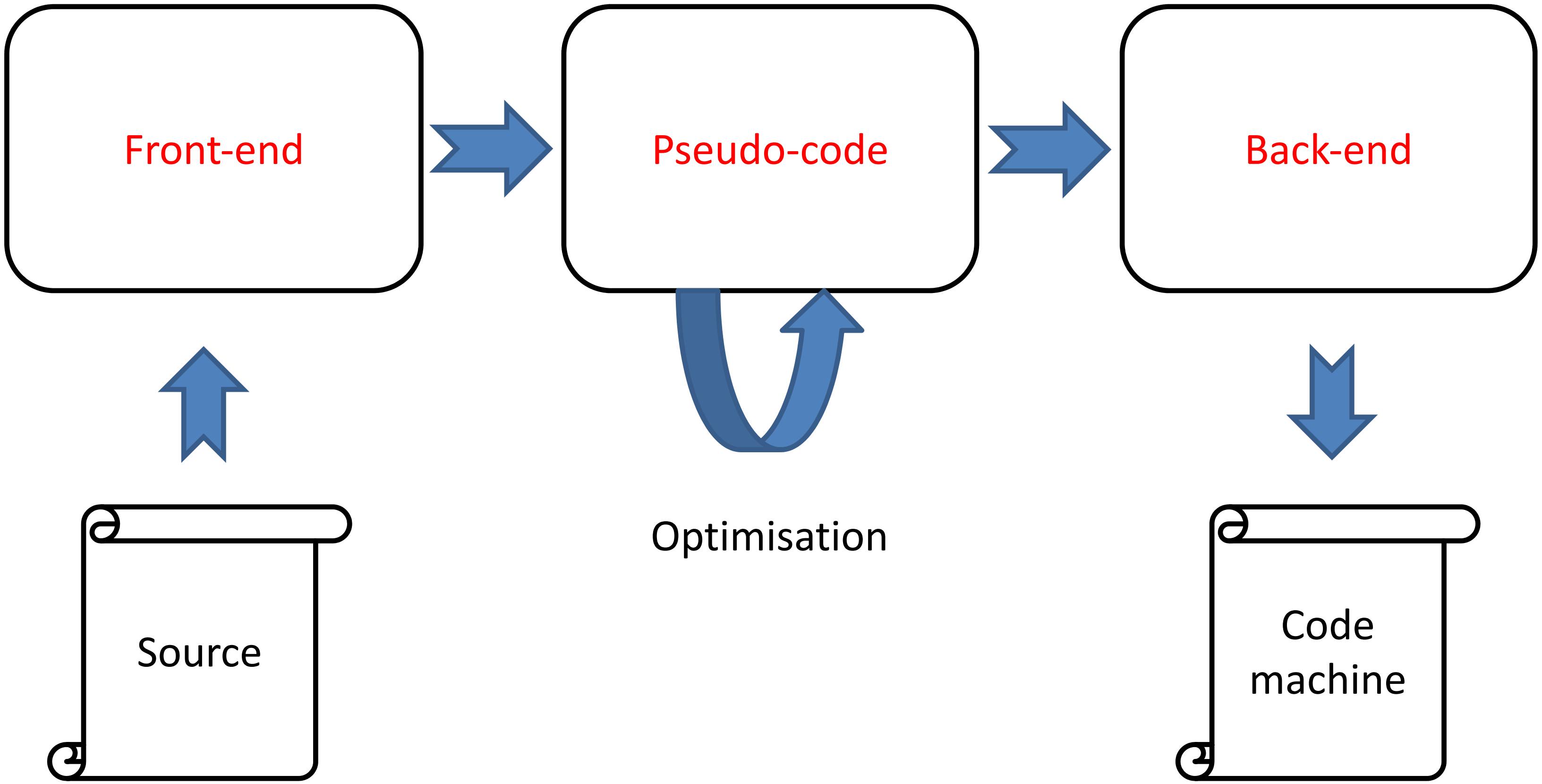
Optimisation de code

Christophe Alias

Contexte


Front-end
Pseudo-code
Back-end
Source
Optimisation
Code
machine

Objectifs

- La traduction dirigée par la syntaxe produit un code de mauvaise qualité
- On cherche à simplifier le code pour réduire sa complexité

Quelles simplifications?

- Factorisation
- Elimination de copies
- Propagation de constantes
- Elimination de code mort
- Réduction de force
- …

```
int main() {
  struct { int re; int im; } s;
  s.im = 0;
  while(s.im<10)
    s.im = s.im + 1;
  return s.im;
}
```

$\rho(s) = t1$

[[s.im]]_t5,t6:

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]

t8 = 1
t2 = t6 + t8

[[s.im]]_t3,t4:

t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]

[t3] = t2

Flot de données

- Dépendances producteur/consommateur

  t = …

  .. = … t ..

- Permet de raisonner sur le calcul

ρ(s) = t1

[[s.im]]_t5,t6

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8

[[s.im]]_t3,t4

t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

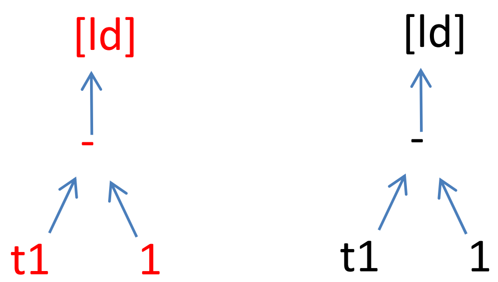
Flot de données

- Dépendances producteur/consommateur

  t = …

  .. = … t ..

- Permet de raisonner sur le calcul



ρ(s) = t1

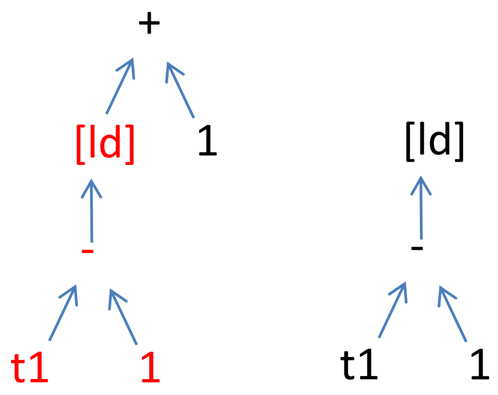
[[s.im]]_t5,t6

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8

[[s.im]]_t3,t4

t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

Exemple




```
int main() {
  struct { int re; int im; } s;
  s.im = 0;
  while(s.im<10)
    s.im = s.im + 1;
  return s.im;
}
```

ρ(s) = t1

[[s.im]]_t5,t6

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]

t8 = 1
t2 = t6 + t8

[[s.im]]_t3,t4

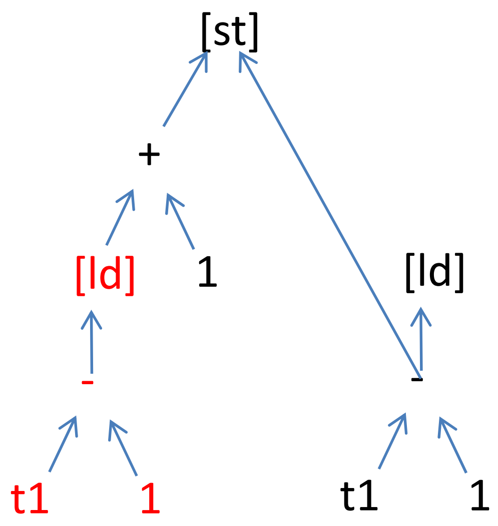
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]

[t3] = t2

Exemple

```
int main() {
  struct { int re; int im; } s;
  s.im = 0;
  while(s.im<10)
    s.im = s.im + 1;
  return s.im;
}
```

$\rho(s) = t1$



[[s.im]]_t5,t6
- t7 = t1
- t10 = 1
- t5 = t7 – t10
- t6 = [t5]

t8 = 1

t2 = t6 + t8

[[s.im]]_t3,t4
- t9 = t1
- t11 = 1
- t3 = t9 – t11
- t4 = [t3]

[t3] = t2

Complexité

```
s.im = 0
while(s.im<10)
  s.im = s.im + 1;
return s.im;
```

Quelle est la définition de **s.im**?

Complexité

```
s.im = 0
while(s.im<10)
  s.im = s.im + 1;
 return s.im;
```

Quelle est la définition de **s.im**?

Le flot de données est incalculable en général!

Stratégies

- Optimisations locales:
  - On divise le programme en parties où le flot de données est calculable
  - On optimise chaque partie indépendamment
  - Flot de données exact $\Rightarrow$ optimisations précises
  - Somme des optimisations locales $\neq$ optimisation globale

Stratégies

- Optimisations locales:
  - On divise le programme en parties où le flot de données est calculable
  - On optimise chaque partie indépendamment
  - Flot de données exact ⇒ optimisations précises
  - Somme des optimisations locales ≠ optimisation globale
- Optimisation globale:
  - On surapproxime le flot de données
  - On optimise d'une traite tout le programme
  - Vue globale ⇒ davantage d'opportunités
  - L'approximation limite les opportunités

Plan

- Optimisations locales
  - Flot de données local
- Optimisations globales
  - Flot de données global
  - Forme SSA

Bloc de base

Plus grande suite d'instructions:

- Sans label (sauf la première instruction)
- Sans saut (sauf la dernière instruction)

```
f:
  alloc 0
  t1 = [$bp+2]
  t2 = 0
loop:
  t3 = 2
  t2 = t2 + t3
  cjump t2<t1 --> loop
end:
  t4 = t2
  ret 1
```

```
int f(int n) {
  int i;
  i = 0
  while(i<n)
    i = i + 2;
  return i;
}
```

$\rho(n) = t1$

$\rho(i) = t2$

Bloc de base

Plus grande suite d'instructions:

- Sans label (sauf la première instruction)
- Sans saut (sauf la dernière instruction)

```
f:
  alloc 0
  t1 = [$bp+2]
  t2 = 0
loop:
  t3 = 2
  t2 = t2 + t3
  cjump t2<t1 --> loop
end:
  t4 = t2
  ret 1
```

```
int f(int n) {
  int i;
  i = 0
  while(i<n)
    i = i + 2;
  return i;
}
```

$\rho(n) = t1$

$\rho(i) = t2$

Graphe de flot de contrôle

Graphe orienté:

- Noeuds: blocs de base
- B1 $\rightarrow$ B2 ssi B2 suit immédiatement B1 dans une exécution

```
f:
  alloc 0
  t1 = [$bp+2]
  t2 = 0
```

```
loop:
  t3 = 2
  t2 = t2 + t3
  cjump t2<t1 --> loop
```

```
end:
  t4 = t2
  ret 1
```

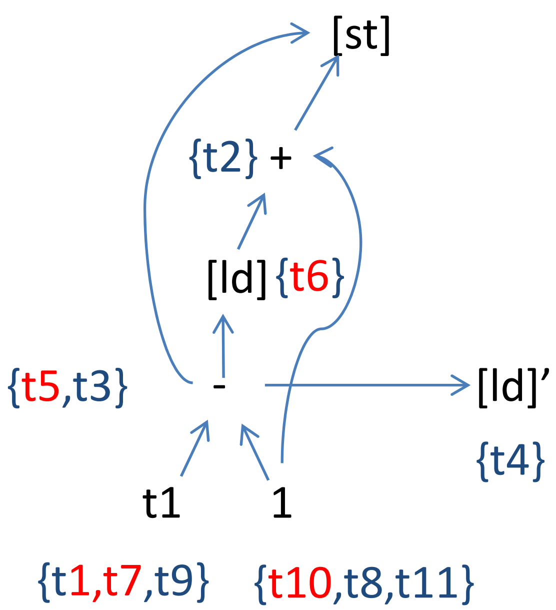
```
int f(int n) {
  int i;
  i = 0
  while(i<n)
    i = i + 2;
  return i;
}
```

$\rho(n) = t1$

$\rho(i) = t2$

Flot de données local

On exécute le bloc de base et on construit le flot de données au fûr et à mesure.



t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

Exemple

t	N(t)
t1	"t1"
t7	"t1"

→ t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

t1

{t1,t7}

Exemple

t	N(t)
t1	“t1”
t7	“t1”
1	“1”
t10	“1”

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

t1   1

{t1,t7}   {t10}

Exemple

{t5} -

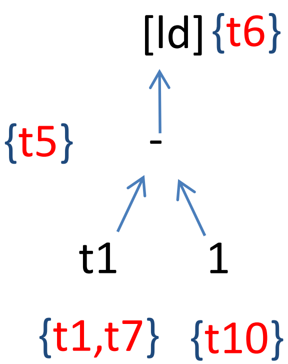
t1 1

{t1,t7} {t10}

t	N(t)
t1	“t1”
t7	“t1”
1	“1”
t10	“1”
t5	“-”

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

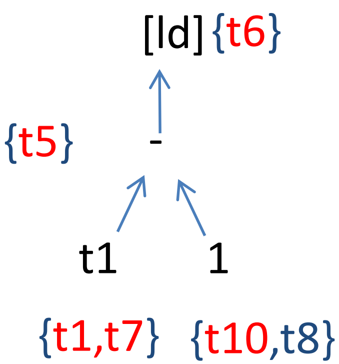
Exemple



t	N(t)
t1	"t1"
t7	"t1"
1	"1"
t10	"1"
t5	"-"
t6	"[Id]"

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

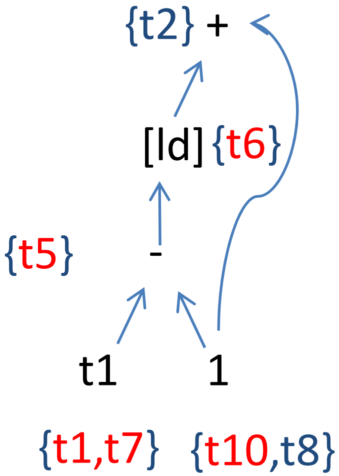
Exemple



t	N(t)
t1	“t1”
t7	“t1”
1	“1”
t10	“1”
t5	“-”
t6	“[Id]’”
t8	“1”

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

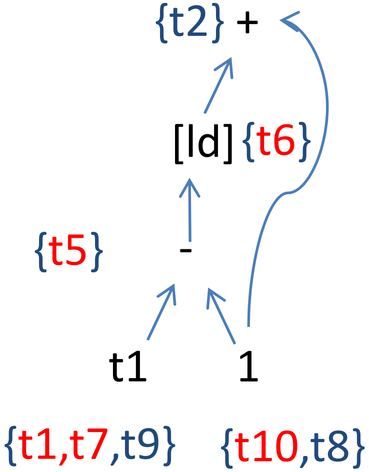
Exemple



t	N(t)
t1	“t1”
t7	“t1”
1	“1”
t10	“1”
t5	“-”
t6	“[ld]’”
t8	“1”
t2	“+”

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

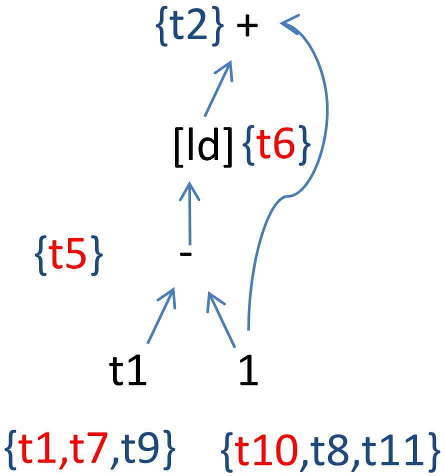
Exemple



t	N(t)
t1	“t1”
t7	“t1”
1	“1”
t10	“1”
t5	“-”
t6	“[ld]’”
t8	“1”
t2	“+”
t9	“t1”

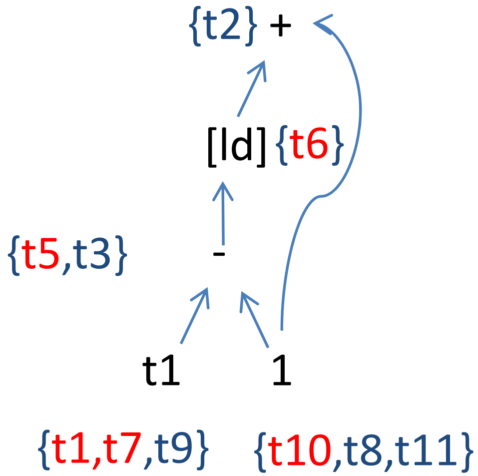
t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

Exemple



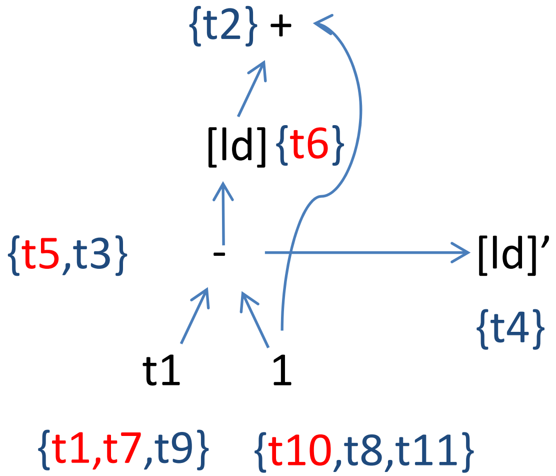
t	N(t)	t7 = t1
t1	“t1”	t10 = 1
t7	“t1”	t5 = t7 – t10
1	“1”	t6 = [t5]
t10	“1”	t8 = 1
t5	“-”	t2 = t6 + t8
t6	“[ld]’”	t9 = t1
t8	“1”	t11 = 1
t2	“+”	t3 = t9 – t11
t9	“t1”	t4 = [t3]
t11	“1”	[t3] = t2

Exemple



t	N(t)	t7 = t1
t1	“t1”	t10 = 1
t7	“t1”	t5 = t7 – t10
1	“1”	t6 = [t5]
t10	“1”	t8 = 1
t5	“-”	t2 = t6 + t8
t6	“[Id]’”	t9 = t1
t8	“1”	t11 = 1
t2	“+”	→ t3 = t9 – t11
t9	“t1”	t4 = [t3]
t11	“1”	[t3] = t2
t3	“-”	

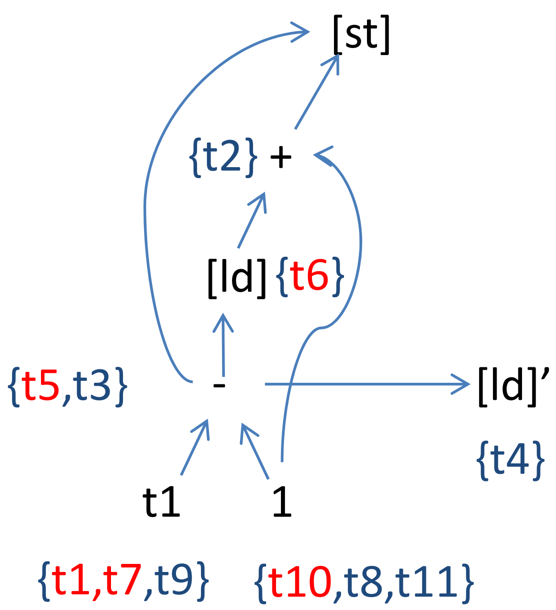
Exemple



t	N(t)
t1	"t1"
t7	"t1"
1	"1"
t10	"1"
t5	"-"
t6	"[Id]'"
t8	"1"
t2	"+"
t9	"t1"
t11	"1"
t3	"-"
t4	"[Id]'"

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
→ t4 = [t3]
[t3] = t2

Exemple



t	N(t)
t1	"t1"
t7	"t1"
1	"1"
t10	"1"
t5	"-"
t6	"[Id]'"
t8	"1"
t2	"+"
t9	"t1"
t11	"1"
t3	"-"
t4	"[Id]'"

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

Algorithme

Pour chaque instruction:

- On créé un noeud N(t) par opérande t
- t = t’
  - N(t) := N(t’)
- t = t’ op t’’
  - Si il existe un noeud n = op(N(t’),N(t’’)) Alors N(t) := n
  - Sinon:
    - créer n = op(N(t’),N(t’’))
    - N(t) := n

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

Algorithme

Pour chaque instruction:

- On créé un noeud N(t) par opérande t
- t = t’
  - N(t) := N(t’)

Elimination de copies

- t = t’ op t’’
  - Si il existe un noeud n = op(N(t’),N(t’’)) Alors N(t) := n
  - Sinon:
    - créer n = op(N(t’),N(t’’))
    - N(t) := n

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

Algorithme

Pour chaque instruction:

- On créé un noeud N(t) par opérande t
- t = t’
  - N(t) := N(t’)

    Elimination de copies
- t = t’ op t’’
  - Si il existe un noeud n = op(N(t’),N(t’’))

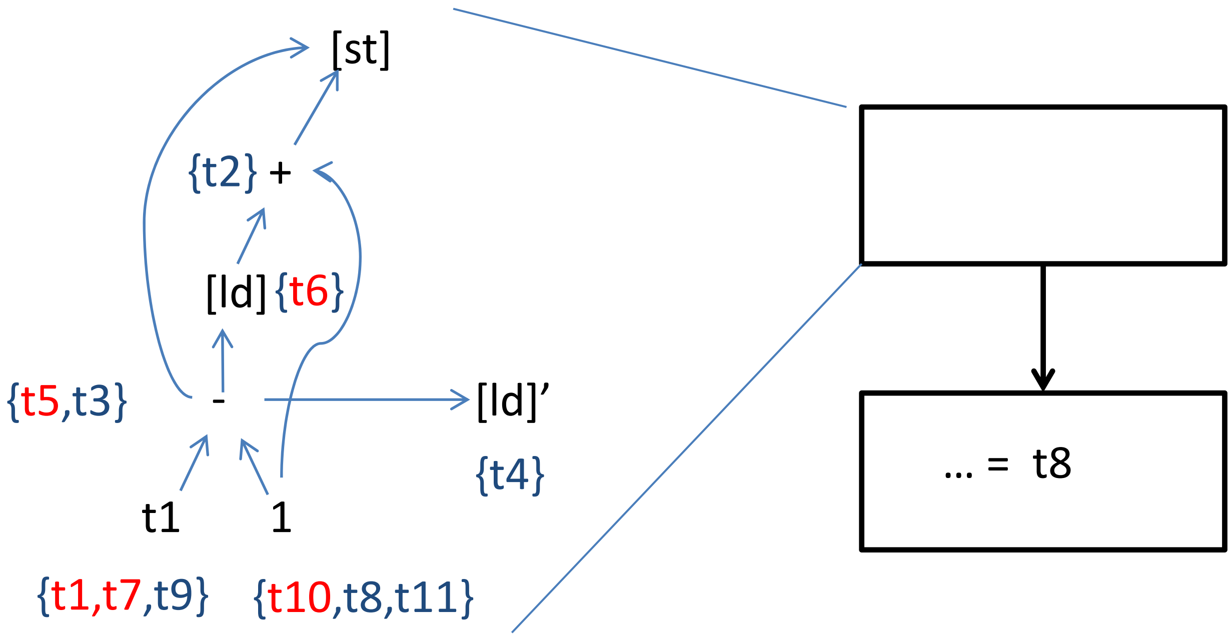
    Alors N(t) := n

    Factorisation
  - Sinon:
    - créer n = op(N(t’),N(t’’))
    - N(t) := n

t7 = t1
t10 = 1
t5 = t7 – t10
t6 = [t5]
t8 = 1
t2 = t6 + t8
t9 = t1
t11 = 1
t3 = t9 – t11
t4 = [t3]
[t3] = t2

Optimisations locales

- On propage les constantes, on effectue les calculs
- Attention à définir chaque temporaire **vivant** en dehors du bloc de base!



Et les accès?

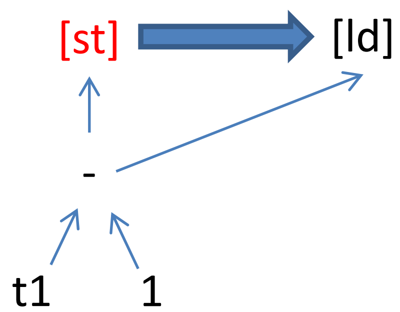
Il manque le flot de données entre les accès:

– Il manque une partie des calculs!

– Il manque des contraintes d'ordre

s.im = ...

... = s.im



Et les accès?

Il manque le flot de données entre les accès:

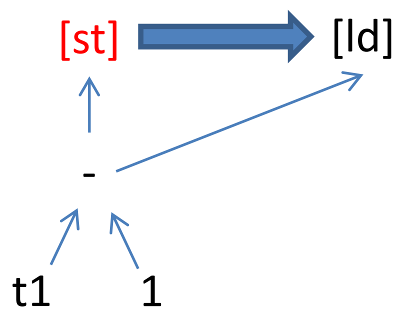
– Il manque une partie des calculs!

– Il manque des contraintes d'ordre

s.im = …

… = s.im

Analyse d'interférence



Complexité

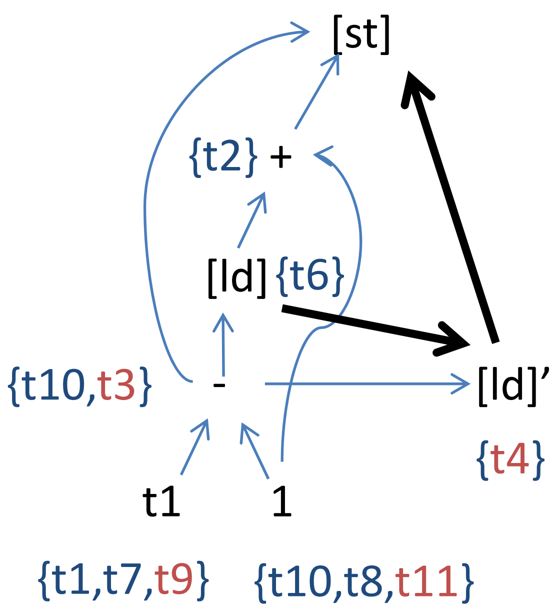
//t1 = P(x1, …, xn)
[t1] = …
//t2 = Q(x1, …, xn)
… = [t2]

Interférence ssi
∃X=(x1…xn): P(X) = Q(X)
(10ème problème de Hilbert)

Le flot de données entre accès est incalculable

Approche conservative

On ajoute des dépendances pour préserver l'ordre original des accés



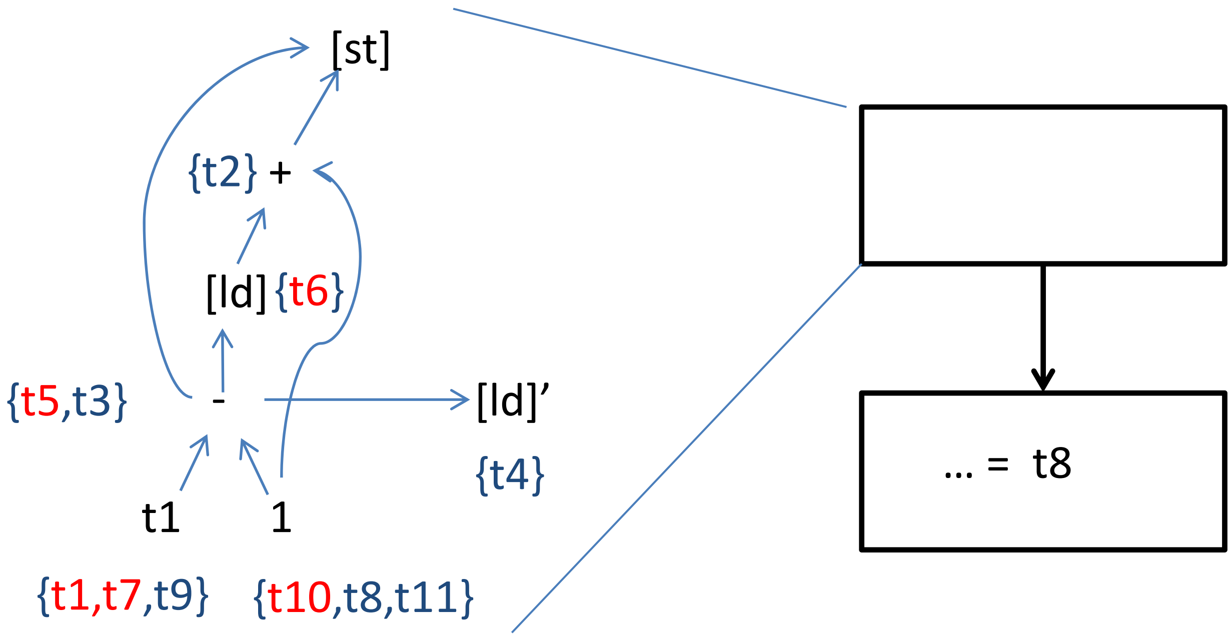

```
      t7 = t1
      t10 = 1
      t5 = t7 – t10
[ld]  t6 = [t5]
      t8 = 1
      t2 = t6 + t8
      t9 = t1
      t11 = 1
      t3 = t9 – t11
[ld'] t4 = [t3]
[st]  [t3] = t2
      stop
```

Plan

- Optimisations locales
  - Flot de données local
- Optimisations globales
  - Flot de données global
  - Forme SSA

Vivacité

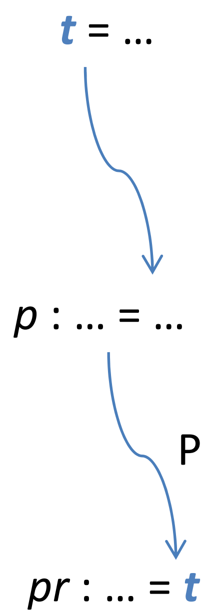
On veut trouver les temporaires vivants en dehors du bloc courant



Vivacité

Un temporaire *t* est vivant en un point d'exécution *p* du programme ssi:

- Il existe un chemin P de *p* vers une lecture de *t*
- Le long de P, *t* n'est pas redéfini



Quizz

Quels temporaires sont vivants :
Juste avant p2?
Juste avant p1?

```
        t1 = [$bp+2]
        t2 = 1
p1:     t3 = t1 + t2
p2:     t2 = 2
        t4 = t3 + t2
        stop
```

Quizz

Quels temporaires sont vivants :

Juste avant p2?

Juste avant p1?

```
        t1 = [$bp+2]

        t2 = 1

p1:     t3 = t1 + t2
{t3}
p2:     t2 = 2

        t4 = t3 + t2

        stop
```

Quizz

Quels temporaires sont vivants :

Juste avant p2?

Juste avant p1?

```
            t1 = [$bp+2]

            t2 = 1
{t1,t2}
p1:         t3 = t1 + t2
{t3}
p2:         t2 = 2

            t4 = t3 + t2

            stop
```

Quizz

Quels temporaires sont vivants :

Juste avant p2?

Juste avant p1?

```
{} ← live in
            t1 = [$bp+2]
{t1}
            t2 = 1
{t1,t2}
p1:         t3 = t1 + t2
{t3}
p2:         t2 = 2
{t2,t3}
            t4 = t3 + t2
{}
            stop
```

Quizz

Quels temporaires sont vivants :

Juste avant p2?

Juste avant p1?

```
{} ← live in
            t1 = [$bp+2]
{t1}
            t2 = 1
{t1,t2}
p1:         t3 = t1 + t2
{t3}
p2:         t2 = 2
{t2,t3}
            t4 = t3 + t2
{} ← live out
```

Quizz

Quels temporaires sont vivants :

Juste avant p2?

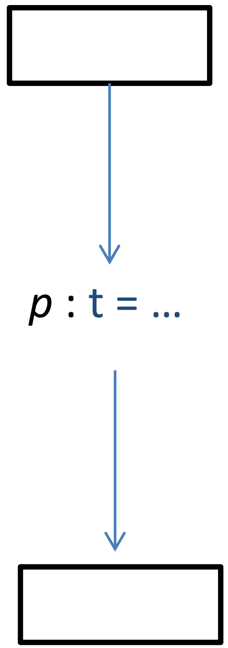
Juste avant p1?

```
{t5} ← live in
            t1 = [$bp+2]
{t1,t5}
            t2 = 1
{t1,t2,t5}
p1:         t3 = t1 + t2
{t3,t5}
p2:         t2 = 2
{t2,t3,t5}
            t4 = t3 + t2
{t5} ← live out
```

Analyse de vivacité

Un temporaire est vivant juste avant p ssi:

- Il est lu par p
  ou
- Il est vivant juste après p
- Il n'est pas défini par p

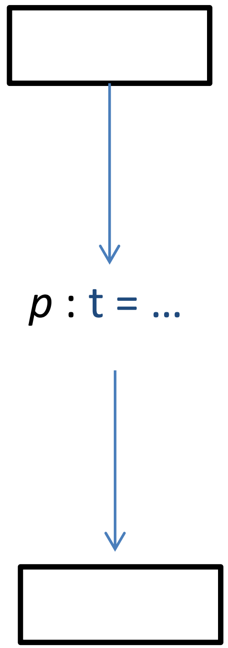



```
         t1 = [$bp+2]
p:       t2 = 1
{t1,t2}
         t3 = t1 + t2
```

Analyse de vivacité

Un temporaire est vivant juste avant p ssi:

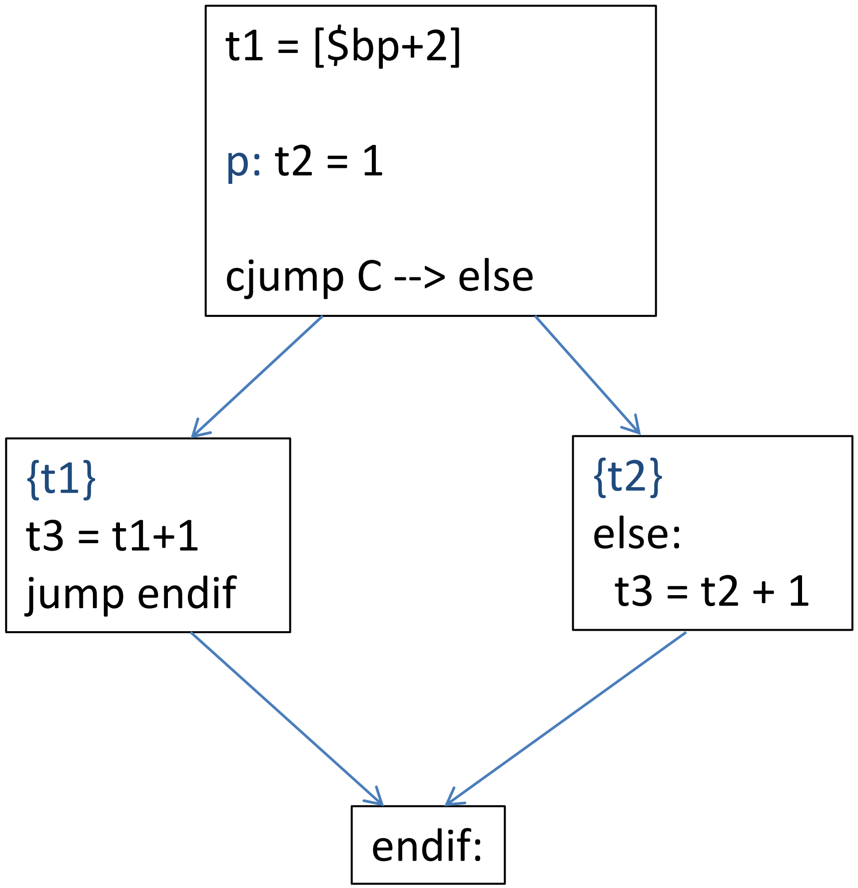
- Il est lu par p
  ou
- Il est vivant juste après p
- Il n'est pas défini par p




```
             t1 = [$bp+2]
{t1}
p:           t2 = 1
{t1,t2}
             t3 = t1 + t2
```

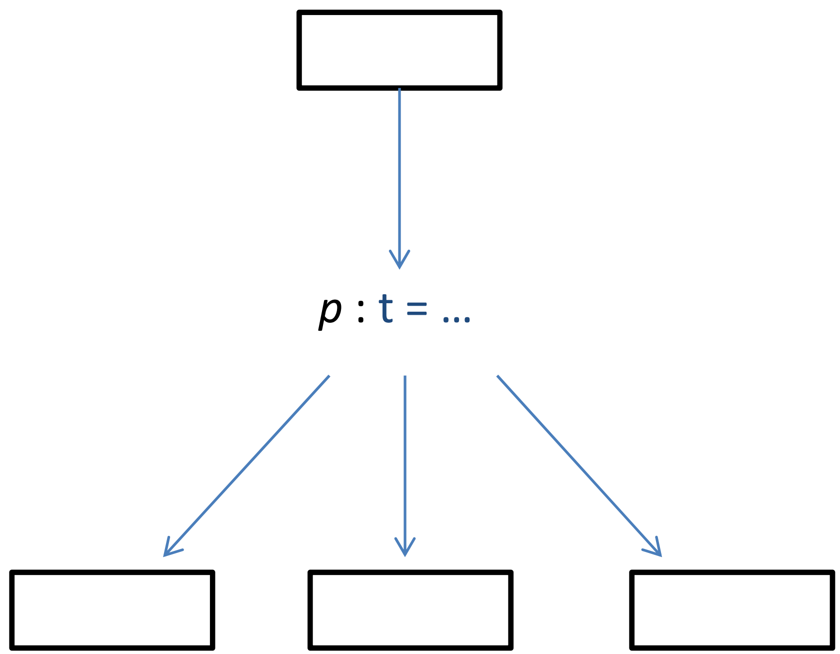
Points de jonction

Quels temporaires sont vivants juste après p?



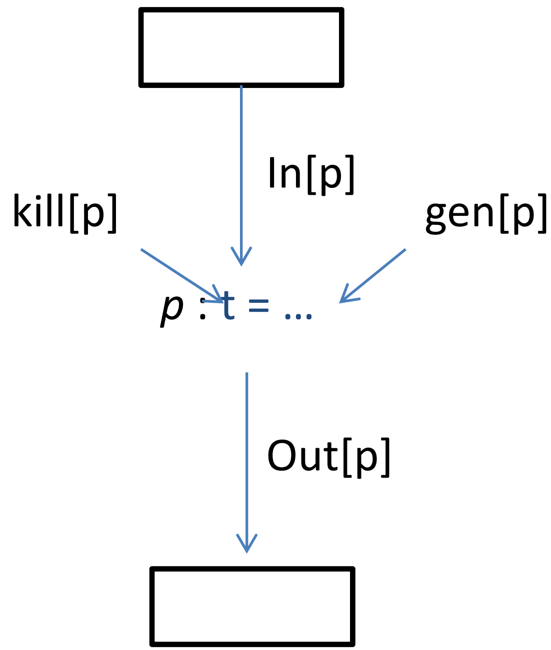
Points de jonction

Un temporaire peut être vivant juste après p s'il est est vivant juste avant l'un des successeurs de p.



Mise en équation

- gen[p]: temporaires lus par p
- kill[p]: temporaire écrit par p
- In[p]: vivacité juste avant p
- Out[p]: vivacité juste après p



t1 = [$bp+2]

{t1}

t2 = 1

{t1,t2}

t3 = t1 + t2

Mise en équation

Un temporaire est vivant juste avant p ssi:

- Il est lu par p

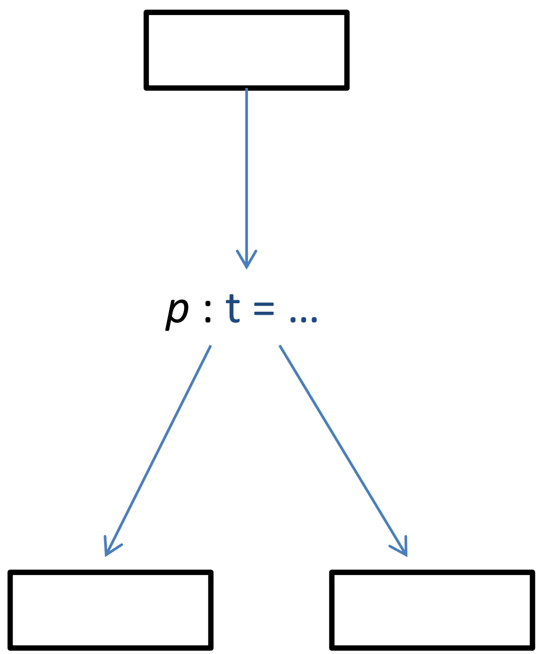
  ou
- Il est vivant juste après p
- Il n'est pas redéfini par p

	t1 = [$bp+2]
{t1}	
	t2 = 1
{t1,t2}	
	t3 = t1 + t2

In[p] = (Out[p] \ kill[p]) U gen[p]

Equations de flot de données

Un temporaire peut être vivant juste après p s'il est est vivant juste avant l'un des successeurs de p.



$$In[p] = (Out[p] \setminus kill[p]) \cup gen[p]$$

$$Out[p] = \cup_{s \in succ(p)} In[s]$$

Résolution

- $X = (In(p_1),Out(p_1), \ldots, In(p_n),Out(p_n))$
- On résout:

$$X = F(X)$$

Résolution

- $X = (In(p_1),Out(p_1), \ldots, In(p_n),Out(p_n))$
- On résout:

$$X = F(X)$$

- Sur quelle structure F opère?
- Propriétés de F?
- Quel point fixe?
- Comment le trouver?

F, structure

- T: temporaires, F opère sur $(2^T)^{2n}$

$$F: (2^T)^{2n} \longrightarrow (2^T)^{2n}$$

F, structure

- T: temporaires, F opère sur $(2^T)^{2n}$

$$F: (2^T)^{2n} \longrightarrow (2^T)^{2n}$$

- Opérateur:

$$X \sqcup X' \Leftrightarrow (X_1 \cup X'_1, \ldots, X_{2n} \cup X'_{2n})$$

F, structure

- T: temporaires, F opère sur $(2^T)^{2n}$

$$F: (2^T)^{2n} \longrightarrow (2^T)^{2n}$$

- Opérateur:

$$X \sqcup X' \Leftrightarrow (X_1 \cup X'_1, \ldots, X_{2n} \cup X'_{2n})$$

- Neutre:

$$\perp = (\emptyset)^{2n} \quad X \sqcup \perp = \perp \sqcup X = X \quad \forall X \in (2^T)^{2n}$$

F, structure

- T: temporaires, F opère sur $(2^T)^{2n}$

  F: $(2^T)^{2n} \longrightarrow (2^T)^{2n}$

- Opérateur:

$$X \sqcup X' \Leftrightarrow (X_1 \cup X'_1, \ldots, X_{2n} \cup X'_{2n})$$

- Neutre:

$$\perp = (\emptyset)^{2n} \quad X \sqcup \perp = \perp \sqcup X = X \quad \forall X \in (2^T)^{2n}$$

- Ordre:

$$X \sqsubseteq X' \Leftrightarrow X_1 \subseteq X'_1, \ldots, X_{2n} \subseteq X'_{2n}$$

F, structure

- T: temporaires, F opère sur $(2^T)^{2n}$

$$F: (2^T)^{2n} \longrightarrow (2^T)^{2n}$$

- Opérateur:

$$X \sqcup X' \Leftrightarrow (X_1 \cup X_1', \ldots, X_{2n} \cup X_{2n}')$$

- Neutre:

$$\perp = (\emptyset)^{2n} \quad X \sqcup \perp = \perp \sqcup X = X \quad \forall X \in (2^T)^{2n}$$

- Ordre:

$$X \sqsubseteq X' \Leftrightarrow X_1 \subseteq X_1', \ldots, X_{2n} \subseteq X_{2n}'$$

- Plus petit élément:

$$\perp = (\emptyset)^{2n} \qquad \perp \sqsubseteq X \quad \forall X \in (2^T)^{2n}$$

F est croissante

- Soient
  $X = (In(p_1), Out(p_1), \ldots, In(p_n), Out(p_n))$
  $X' = (In'(p_1), Out'(p_1), \ldots, In'(p_n), Out'(p_n))$
  avec $X \sqsubseteq X'$

F est croissante

- Soient
  $X = (In(p_1), Out(p_1), \ldots, In(p_n), Out(p_n))$
  $X' = (In'(p_1), Out'(p_1), \ldots, In'(p_n), Out'(p_n))$
  avec $X \sqsubseteq X'$
- Alors, pour tout p:
  - $(Out(p) \setminus kill(p)) \cup gen(p) \subseteq (Out'(p) \setminus kill(p)) \cup gen(p)$
  - $\bigcup_{s \in succ(p)} In[s] \subseteq \bigcup_{s \in succ(p)} In'[s]$

F est croissante

- Soient
  $X = (In(p_1), Out(p_1), \ldots, In(p_n), Out(p_n))$
  $X' = (In'(p_1), Out'(p_1), \ldots, In'(p_n), Out'(p_n))$
  avec $X \sqsubseteq X'$
- Alors, pour tout p:
  - $(Out(p) \setminus kill(p)) \cup gen(p) \subseteq (Out'(p) \setminus kill(p)) \cup gen(p)$
  - $\bigcup_{s \in succ(p)} In[s] \subseteq \bigcup_{s \in succ(p)} In'[s]$
- Ainsi: $F(X) \sqsubseteq F(X')$

Plan

- Sur quelle structure F opère?
- Propriétés de F?
- Quel point fixe?
- Comment le trouver?

Quel point fixe?

{} ← live-in

    t1 = [bp+2]

{t1}

    t2 = 1

{t1,t2}

    t3 = t1+t2

{} ← live-out

Solution X

Quel point fixe?

T ← live-in

t1 = [bp+2]

{t1} **U T**

t2 = 1

{t1,t2} **U T**

t3 = t1+t2

T ← live-out

Solution X_T

Quel point fixe?

T $\leftarrow$ live-in

t1 = [bp+2]

{t1} **U T**

t2 = 1

{t1,t2} **U T**

t3 = t1+t2

T $\leftarrow$ live-out

$$X \sqsubseteq X_T$$

Solution X_T

Quel point fixe?

T $\leftarrow$ live-in

t1 = [bp+2]

{t1} **U T**

t2 = 1

{t1,t2} **U T**

t3 = t1+t2

T $\leftarrow$ live-out

Solution X_T

$X \sqsubseteq X_T$

On cherche **le plus petit point fixe de F**

Plan

- Sur quelle structure F opère?
- Propriétés de F?
- Quel point fixe?
- Comment le trouver?

Résolution

Si T est fini et

$$F : ((2^T)^{2n}, \sqsubseteq, \sqcup, \perp) \rightarrow ((2^T)^{2n}, \sqsubseteq, \sqcup, \perp)$$

est croissante

Résolution

Si T est fini et

$$F : ((2^T)^{2n}, \sqsubseteq, \sqcup, \perp) \rightarrow ((2^T)^{2n}, \sqsubseteq, \sqcup, \perp)$$

est croissante

Alors F admet un plus petit point fixe:

$$\mathrm{lfp}(F) = \lim_{n \to +\infty} F^n(\perp)$$

Résolution

Si T est fini et

$$F : ((2^T)^{2n}, \sqsubseteq, \sqcup, \perp) \rightarrow ((2^T)^{2n}, \sqsubseteq, \sqcup, \perp)$$

est croissante

Alors F admet un plus petit point fixe:

$$\mathrm{lfp}(F) = \lim_{n \to +\infty} F^n(\perp)$$

La limite existe

C'est un point fixe

C'est le plus petit

La limite existe

- $(F^n(\perp))_n$ est une suite dans un ensemble fini
  Il suffit de montrer qu'elle est croissante:

$$F^n(\perp) \sqsubseteq F^{n+1}(\perp) \quad \forall n \in \mathbb{N}$$

La limite existe

- $(F^n(\perp))_n$ est une suite dans un ensemble fini
  Il suffit de montrer qu'elle est croissante:
  $$F^n(\perp) \sqsubseteq F^{n+1}(\perp) \quad \forall n \in \mathbb{N}$$
- Par récurrence sur n:
  - n=0: $F^0(\perp) = \perp \sqsubseteq F^1(\perp)$

La limite existe

- $(F^n(\perp))_n$ est une suite dans un ensemble fini
  Il suffit de montrer qu'elle est croissante:
  $$F^n(\perp) \sqsubseteq F^{n+1}(\perp) \quad \forall n \in \mathbb{N}$$
- Par récurrence sur n:
  - n=0: $F^0(\perp) = \perp \sqsubseteq F^1(\perp)$
  - HR: $F^n(\perp) \sqsubseteq F^{n+1}(\perp)$
    Comme F est croissante:
    $$F^{n+1}(\perp) \sqsubseteq F^{n+2}(\perp)$$

La limite existe

- $(F^n(\bot))_n$ est une suite dans un ensemble fini
  Il suffit de montrer qu'elle est croissante:
  $$F^n(\bot) \sqsubseteq F^{n+1}(\bot) \quad \forall n \in \mathbb{N}$$
- Par récurrence sur n:
  - n=0: $F^0(\bot) = \bot \sqsubseteq F^1(\bot)$
  - HR: $F^n(\bot) \sqsubseteq F^{n+1}(\bot)$
    Comme F est croissante:
    $$F^{n+1}(\bot) \sqsubseteq F^{n+2}(\bot)$$
- $(F^n(\bot))_n$ est donc stationnaire

C’est un point fixe

- Soit $\ell = \lim_{n \to +\infty} F^n(\perp)$

C’est un point fixe

- Soit $\ell = \lim_{n \to +\infty} F^n(\perp)$
- Comme $(F^n(\perp))_n$ est stationnaire:

$$\exists N : n \geq N \Rightarrow F^n(\perp) = \ell$$

C’est un point fixe

- Soit $\ell = \lim\limits_{n \to +\infty} F^n(\perp)$
- Comme $(F^n(\perp))_n$ est stationnaire:

$$\exists N : n \geq N \Rightarrow F^n(\perp) = \ell$$

- Ainsi:

$$F(\ell) = F(F^N(\perp)) = F^{N+1}(\perp) = \ell$$

C’est le plus petit

- Soit ℓ' un point fixe de F
- On montre que:

$$F^n(\bot) \sqsubseteq \ell' \quad \forall n \in \mathbb{N}$$

C’est le plus petit

- Soit ℓ' un point fixe de F
- On montre que:

$$F^n(\perp) \sqsubseteq \ell' \quad \forall n \in \mathbb{N}$$

- Par récurrence sur n:
  - Cas 0: $F^0(\perp) = \perp \sqsubseteq \ell'$

C’est le plus petit

- Soit ℓ' un point fixe de F
- On montre que:

$$F^n(\bot) \sqsubseteq \ell' \quad \forall n \in \mathbb{N}$$

- Par récurrence sur n:
  - Cas 0: $F^0(\bot) = \bot \sqsubseteq \ell'$
  - HR: $F^n(\bot) \sqsubseteq \ell'$

    F croissante, donc:

$$F(F^n(\bot)) \sqsubseteq F(\ell')$$

$$F^{n+1}(\bot) \sqsubseteq \ell'$$

Algorithme

$$
\begin{array}{l}
X := \bot; \\
\textbf{while } (X \neq F(X)) \\
\qquad X := F(X); \\
\textbf{return } X;
\end{array}
$$

Algorithme

$$
\begin{array}{l}
X := \bot; \\
\textbf{while } (X \neq F(X)) \\
\quad X := F(X); \\
\textbf{return } X;
\end{array}
$$

- In[p] := vide; Out[p] := vide
- Répéter

  pour chaque p:

    In'[p] := (Out[p] \ kill[p]) U gen[p]

    Out'[p] := $\mathrm{U}_{s \in succ(p)}$ In[s]

  Si In = In' et Out = Out' Alors fin

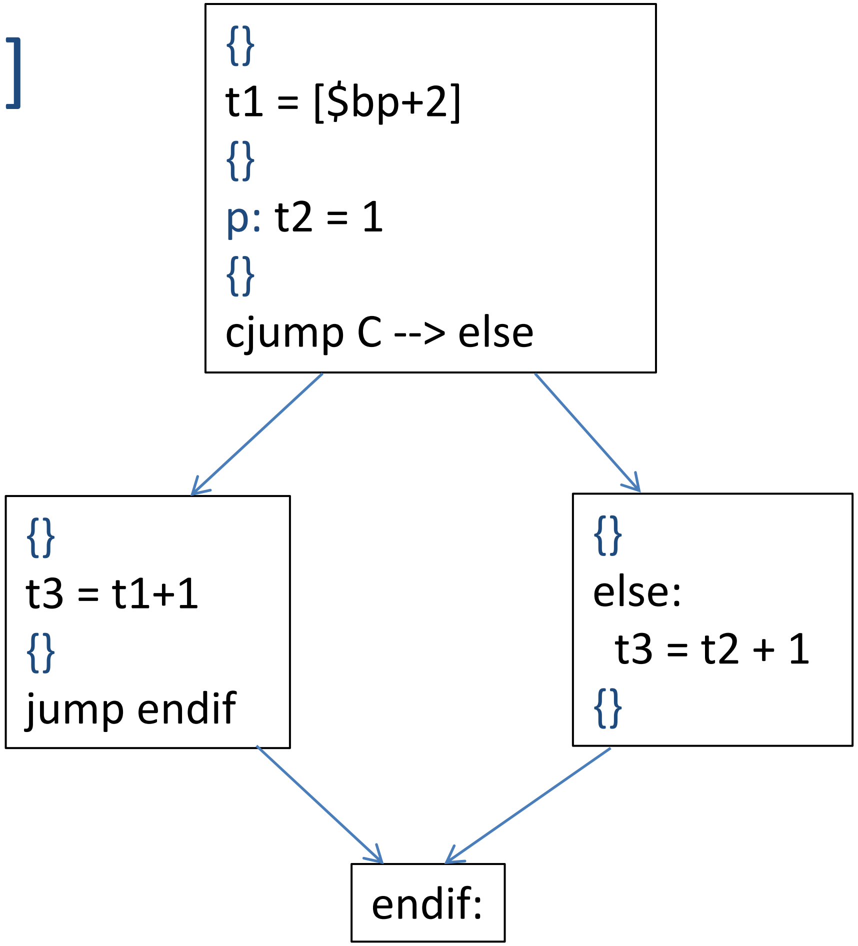
  In := In'

  Out := Out'

Résolution : $\perp$

$$In[p] = (Out[p] \setminus kill[p]) \cup gen[p]$$

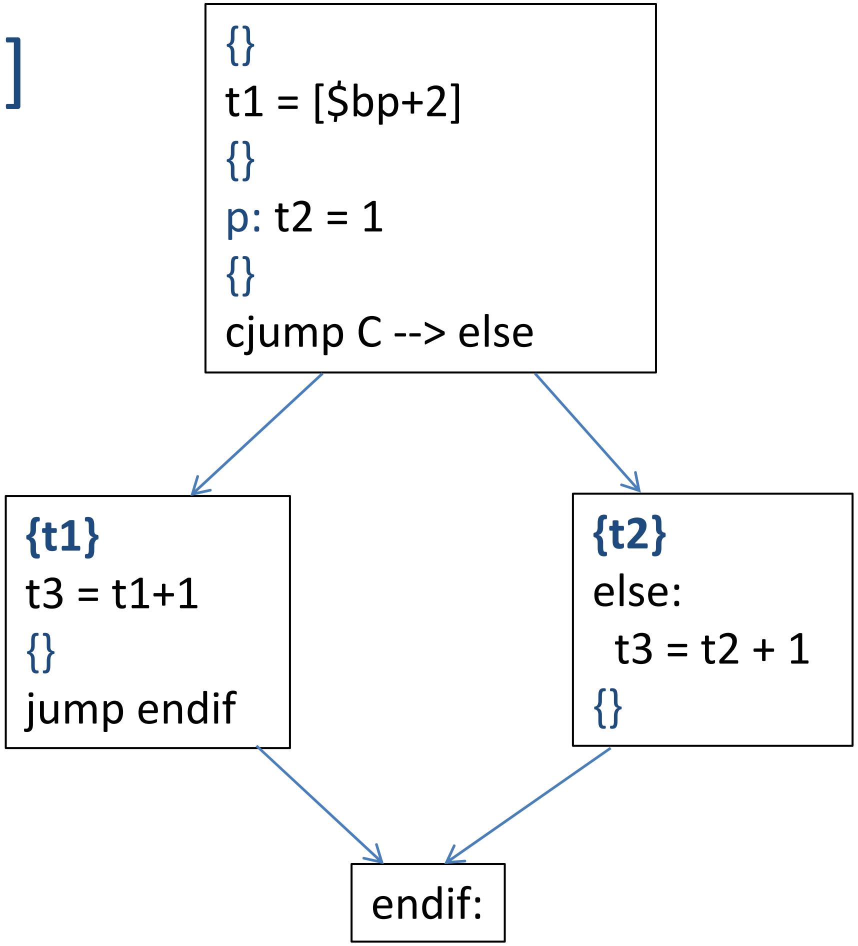
$$Out[p] = \cup_{s \in succ(p)} In[s]$$



Résolution : $F(\perp)$

$\text{In}[p] = (\text{Out}[p] \setminus \text{kill}[p]) \cup \text{gen}[p]$

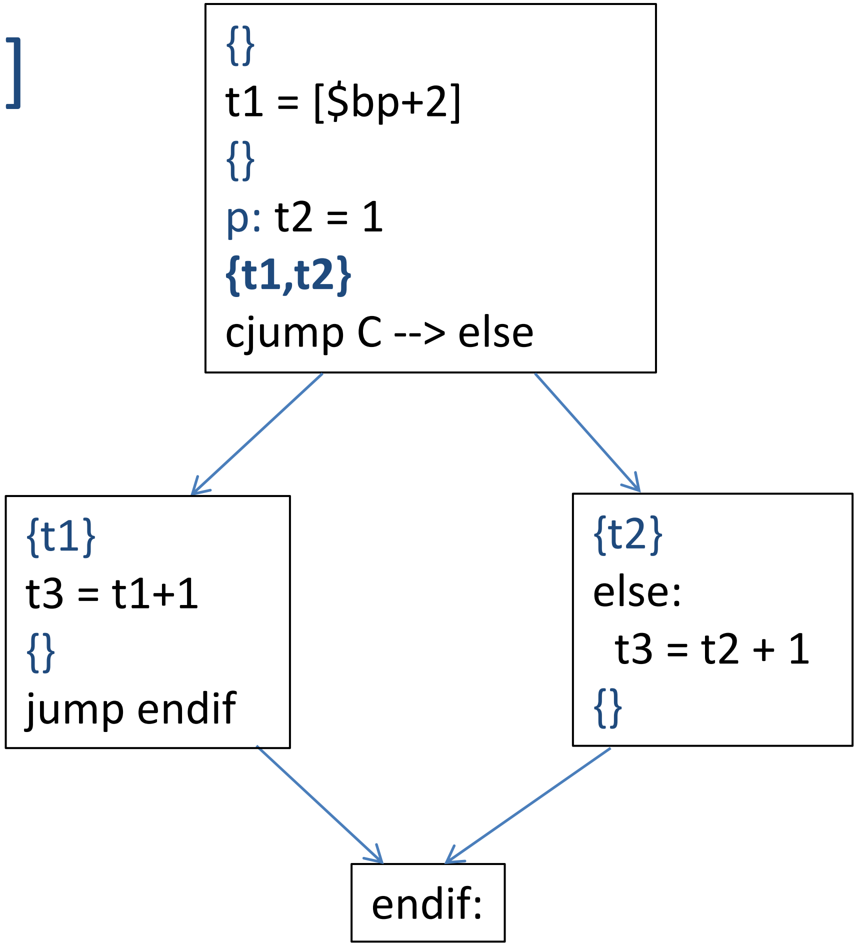
$\text{Out}[p] = \cup_{s \in \text{succ}(p)} \text{In}[s]$



Résolution : $F^2(\perp)$

In[p] = (Out[p] \ kill[p]) U gen[p]

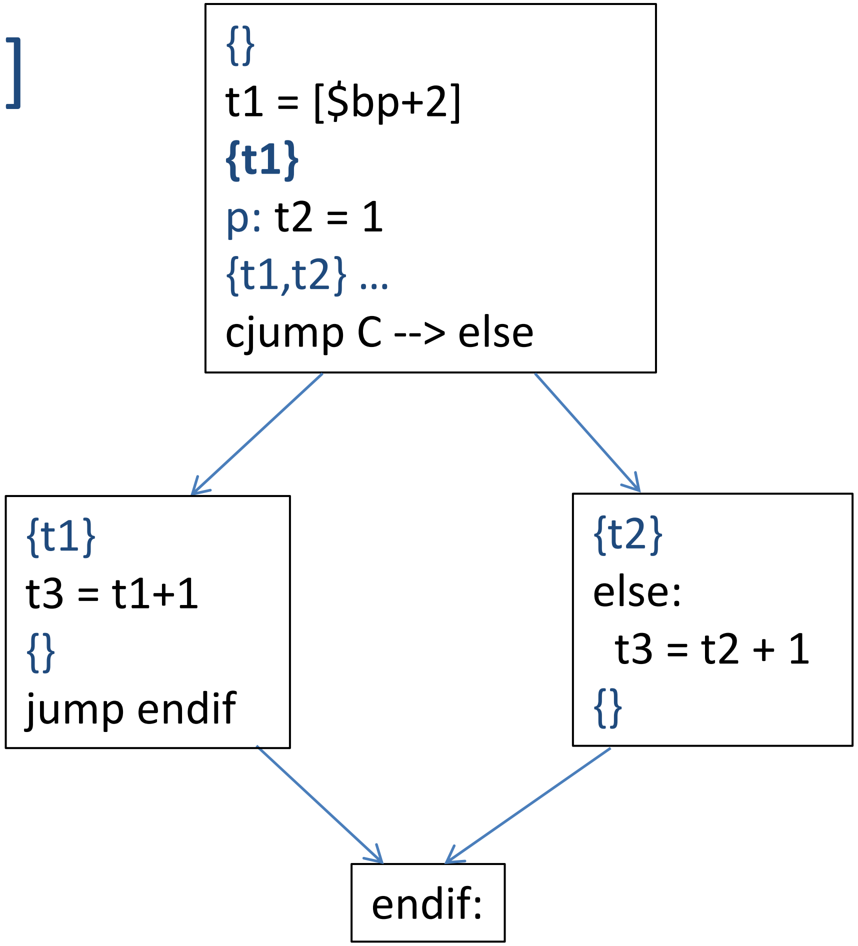
Out[p] = $U_{s \in succ(p)}$ In[s]



Résolution : $F^3(\perp)$

In[p] = (Out[p] \ kill[p]) U gen[p]

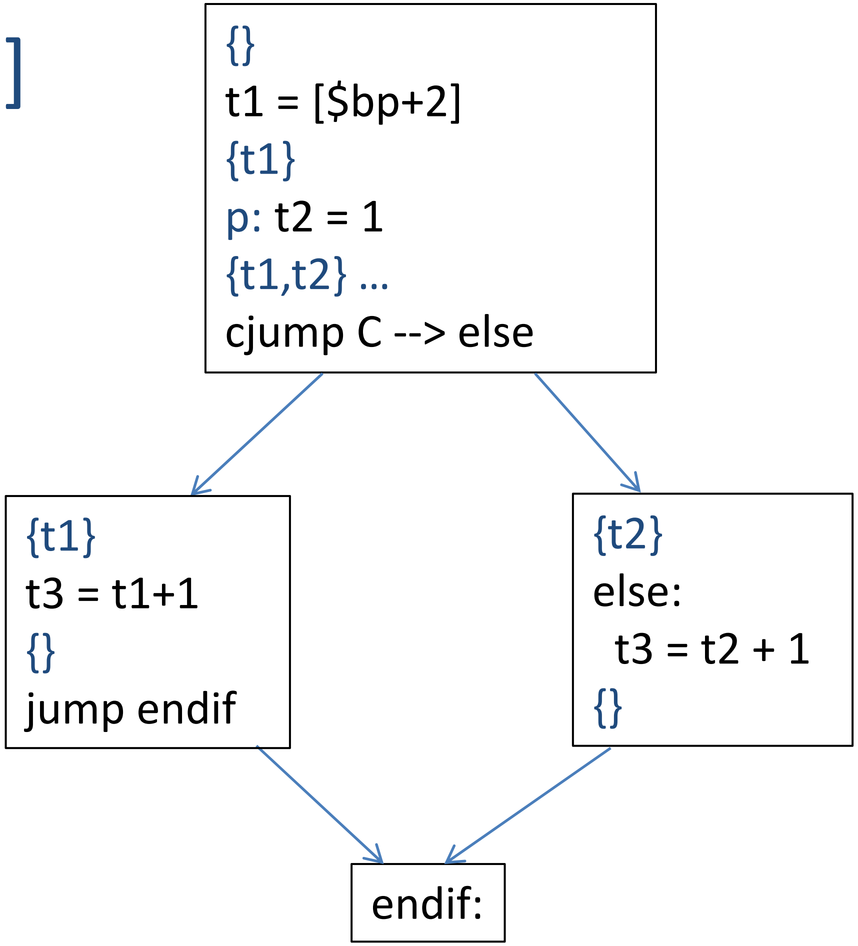
Out[p] = $U_{s \in \text{succ}(p)}$ In[s]



Résolution : $F^4(\perp) = F^3(\perp)$

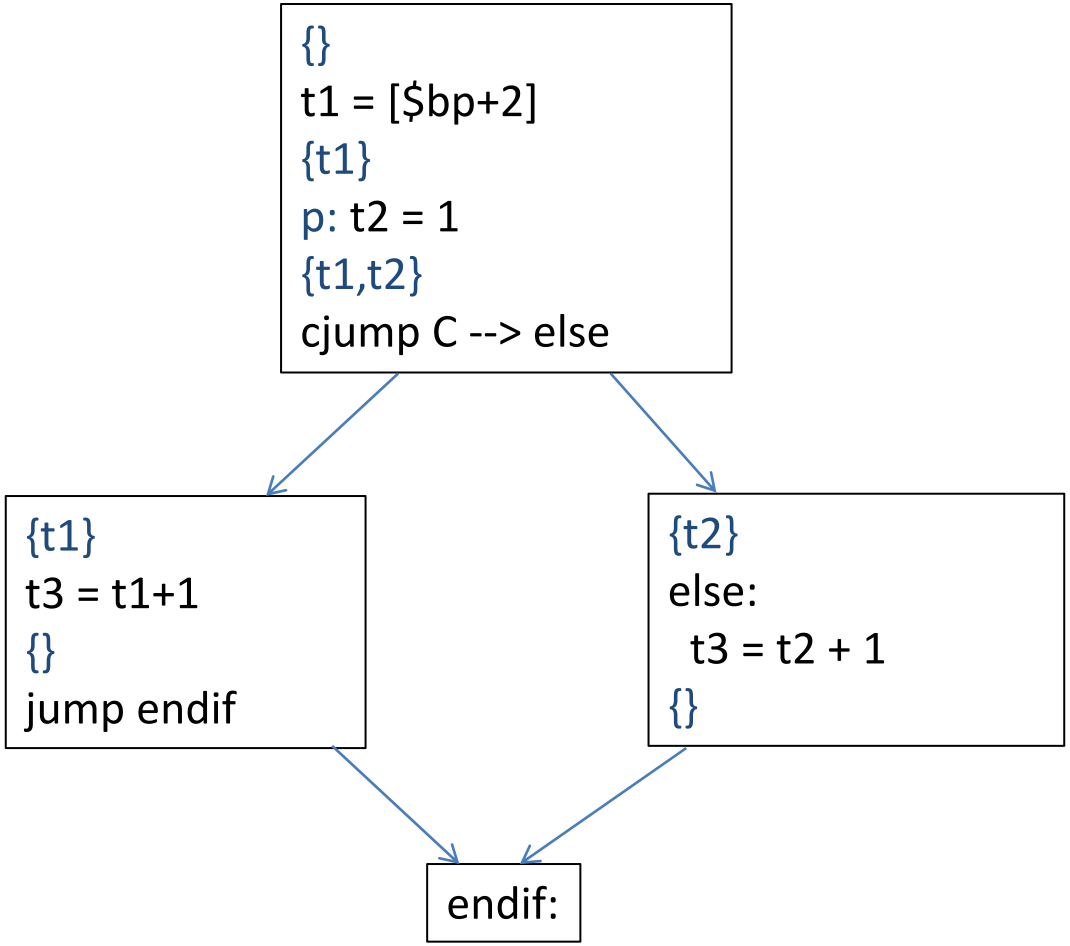
In[p] = (Out[p] \ kill[p]) U gen[p]

Out[p] = $U_{s \in succ(p)}$ In[s]



Inchangé

Solution


{}
t1 = [$bp+2]
{t1}
p: t2 = 1
{t1,t2}
cjump C --> else
{t1}
t3 = t1+1
{}
jump endif
{t2}
else:
t3 = t2 + 1
{}
endif:

Expressions disponibles

```
f:
alloc 0
t1 = [$bp+2]
t2 = 1
t3 = t1 – t2
[t3] = t2
```

```
loop: ...
t4 = t1 – t2
t5 = [t4]
t6 = 10
cjump(t5 < t6) --> loop
```

```
free 0
ret 1
```

Comment optimiser ce programme?

```
void f(int[10] tab) {
  tab[1] = 1
  do
    ...
  while(tab[1] < 10)
}
```

$\rho(tab) = t1$

Expressions disponibles

```
f:
alloc 0
t1 = [$bp+2]
t2 = 1
t3 = t1 – t2
[t3] = t2
```

Pour tout chemin:
t1 – t2 est déjà calculé

```
loop: ...
t4 = t1 – t2
t5 = [t4]
t6 = 10
cjump(t5 < t6) --> loop
```

```
free 0
ret 1
```

Comment optimiser ce programme?

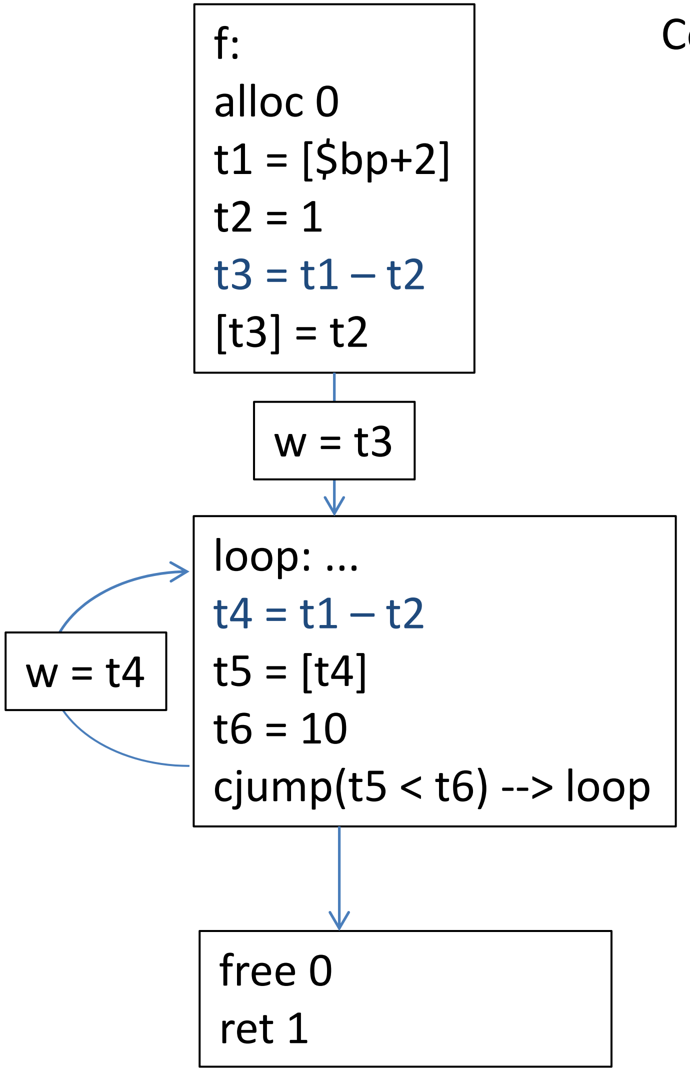
```
void f(int[10] tab) {
  tab[1] = 1
  do
    ...
  while(tab[1] < 10)
}
```

ρ(tab) = t1

Expressions disponibles

Comment optimiser ce programme?

Optimisation




```
void f(int[10] tab) {
  tab[1] = 1
  do
    ...
  while(tab[1] < 10)
}
```

$\rho(tab) = t1$

Expressions disponibles

```
f:
alloc 0
t1 = [$bp+2]
t2 = 1
t3 = t1 – t2
[t3] = t2
```

```
w = t3
```

```
loop: ...
t4 = w
t5 = [t4]
t6 = 10
cjump(t5 < t6) --> loop
```

```
w = t4
```

```
free 0
ret 1
```

Optimisation

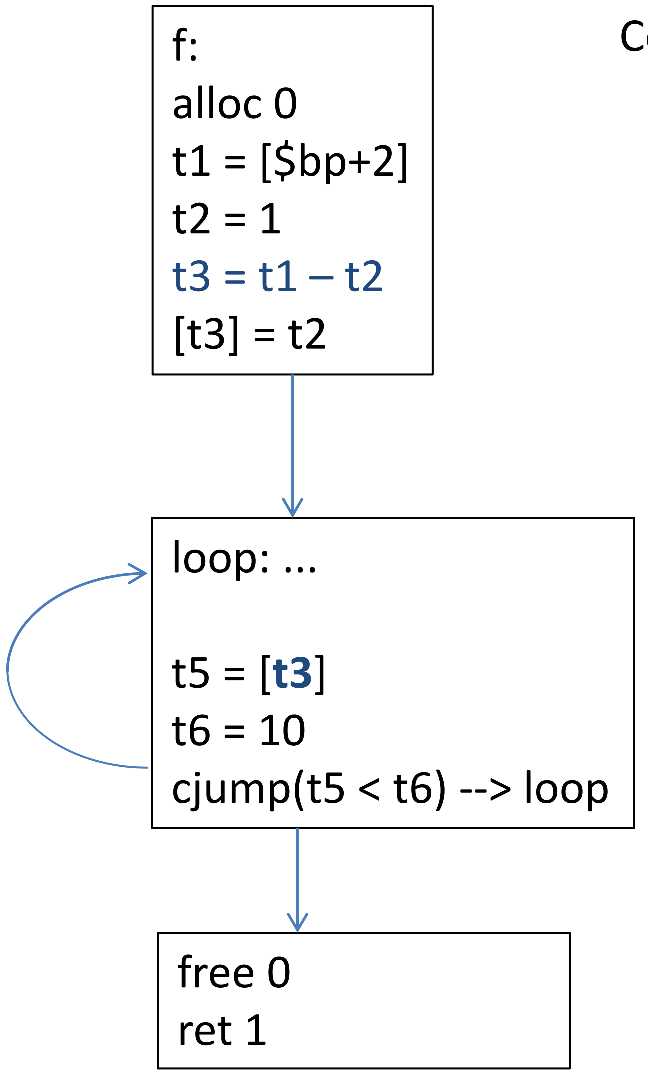
Comment optimiser ce programme?

```
void f(int[10] tab) {
 tab[1] = 1
 do
  ...
 while(tab[1] < 10)
}
```

$\rho(tab) = t1$

Expressions disponibles

Comment optimiser ce programme?




```
void f(int[10] tab) {
  tab[1] = 1
  do
    ...
  while(tab[1] < 10)
}
```

$\rho(\text{tab}) = \text{t1}$

Optimisation

Propagation de copies

Quizz

Donner les expressions disponibles

t4 = t1 – t2

?

t1 = t1 + t2

?

Quizz

Donner les expressions disponibles

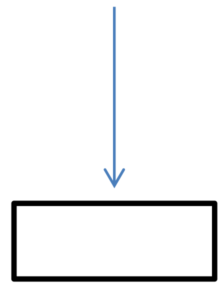
$$
\boxed{
\begin{array}{l}
t4 = t1 - t2 \\
\{\ t1 - t2\ \} \\
t1 = t1 + t2 \\
\{\ \}
\end{array}
}
$$

Expressions disponibles

E est disponible juste après *p*
si *p* ne défini pas un temporaire de E

$t4 = t1 - t2$
$\{\ t1 - t2\ \}$
$t1 = t1 + t2$
$\{\ \}$

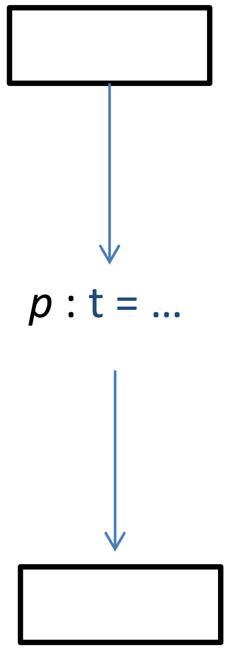
p : t = E



Propagation de l'information

Une expression E est disponible juste après *p* si:

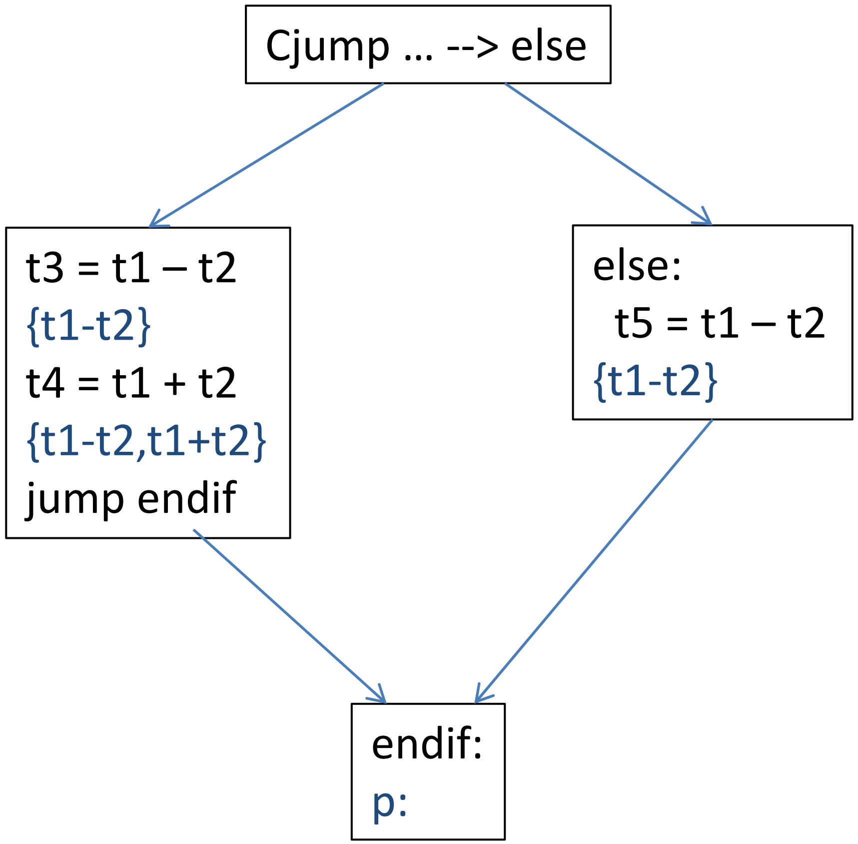
- E est disponible juste avant p
  et
  p ne défini pas un temporaire de E



t4 = t1 – t2
{ t1 – t2 }
t3 = 1
{ t1 – t2}

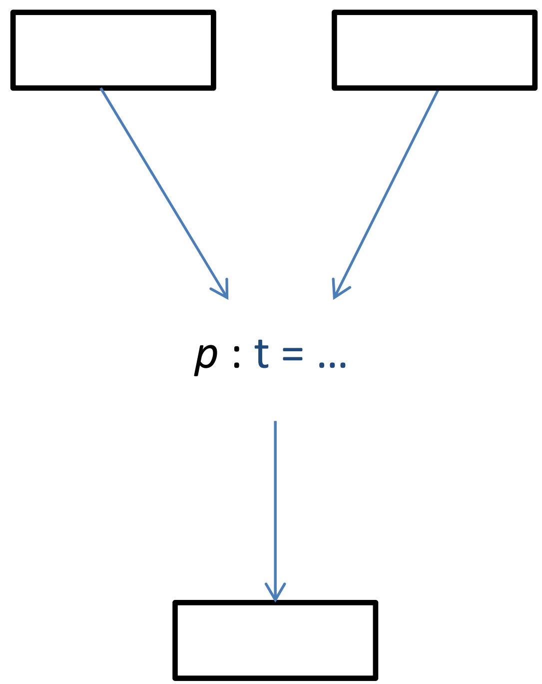
Points de jonction

Quelles expressions sont disponibles juste avant *p*?



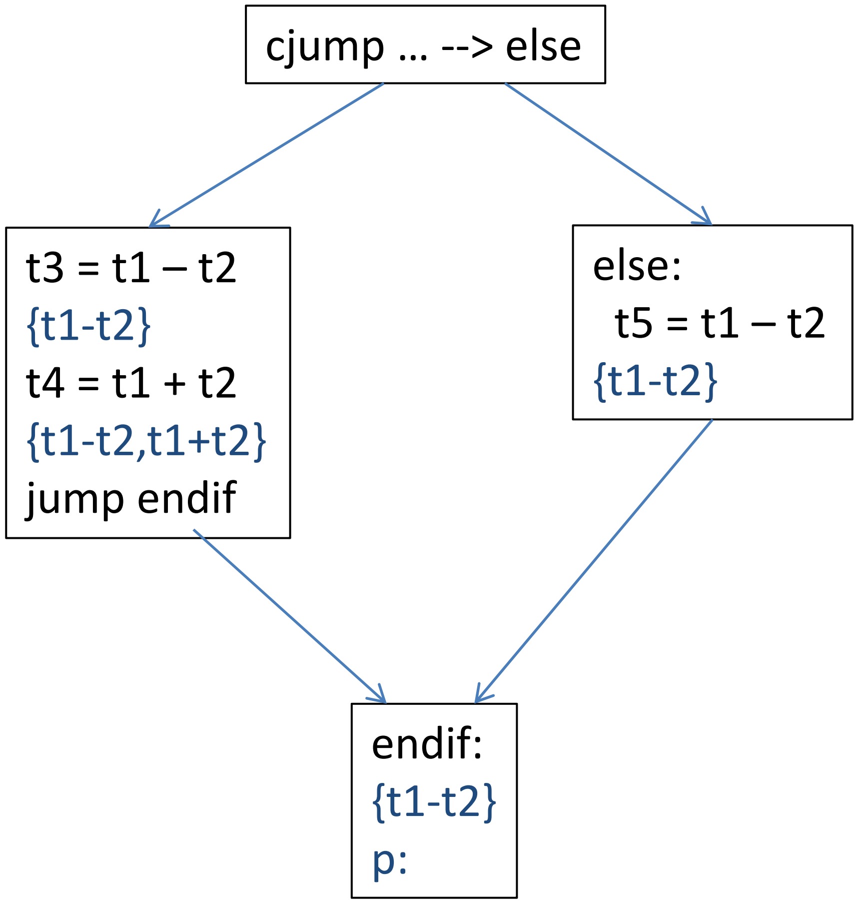
Points de jonction

- Une expression est disponible juste avant *p* si elle disponible juste après chaque prédecesseur de *p*



Points de jonction

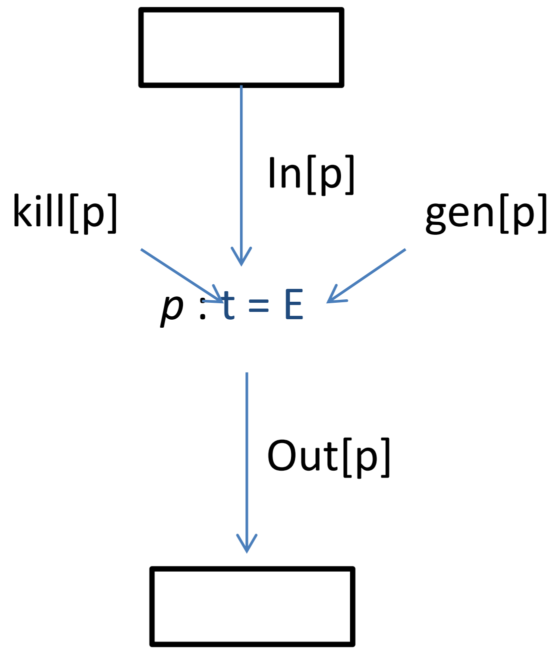
Quelles expressions sont disponibles juste avant *p*?



intersection

Mise en équation

- gen[p]: E
- kill[p]: expressions avec t
- In[p]: disponible juste avant p
- Out[p]: disponible juste après p



t4 = t1 – t2
{ t1 – t2 }
t1 = t1 + t2
{ }

Mise en équation

Une expression E est disponible juste après *p* si:

- *p* utilise E et ne défini pas un temporaire de E

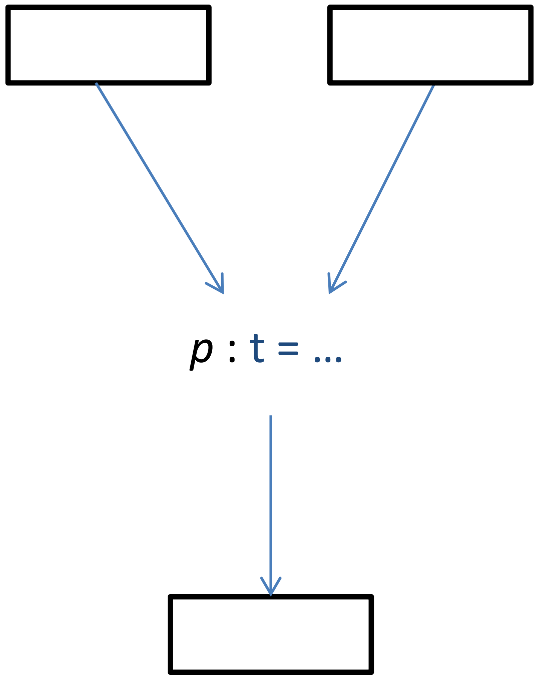
ou

- E est disponible juste avant p
  et
  p ne défini pas un temporaire de E

Out[p] = (In[p] U gen[p]) \ kill[p]

Mise en équation

- Une expression est disponible juste avant *p* si elle disponible juste après chaque prédecesseur de *p*



$$Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$$

$$In[p] = \bigcap_{q \in pred(p)} Out[q]$$

Résolution

$$Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$$

$$In[p] = \bigcap_{q \in pred(p)} Out[q]$$

```
t3 = t1 – t2
{}
[t3] = t2
{}
```

```
loop: ...
{}
t4 = t1 – t2
{}
t5 = [t4]
{}
t6 = 10
{}
cjump(t5 < t6) --> loop
```

Résolution, itération 1

$$Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$$

$$In[p] = \bigcap_{q \in pred(p)} Out[q]$$

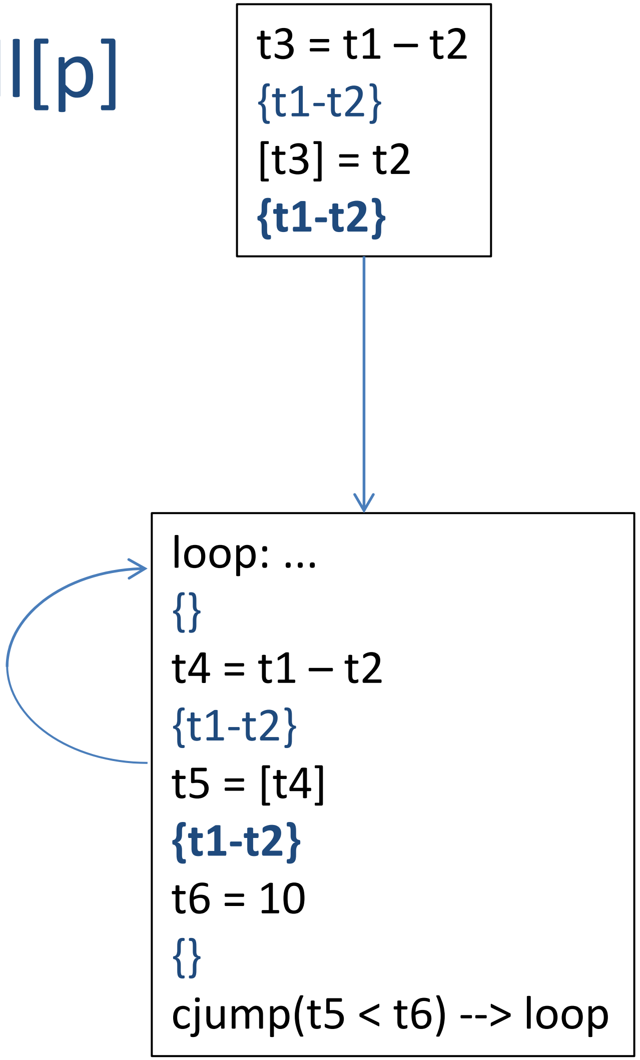
```
t3 = t1 – t2
{t1-t2}
[t3] = t2
{}
```

```
loop: ...
{}
t4 = t1 – t2
{t1-t2}
t5 = [t4]
{}
t6 = 10
{}
cjump(t5 < t6) --> loop
```

Résolution, itération 2

$$Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$$

$$In[p] = \bigcap_{q \in pred(p)} Out[q]$$



Résolution, itération 3

$$Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$$

$$In[p] = \bigcap_{q \in pred(p)} Out[q]$$

t3 = t1 – t2
{t1-t2}
[t3] = t2
{t1-t2}

loop: ...
{}
t4 = t1 – t2
{t1-t2}
t5 = [t4]
{t1-t2}
t6 = 10
{t1-t2}
cjump(t5 < t6) --> loop

Résolution, itération 4

$$Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$$

$$In[p] = \cap_{q \in pred(p)} Out[q]$$

t3 = t1 – t2
{t1-t2}
[t3] = t2
{t1-t2}

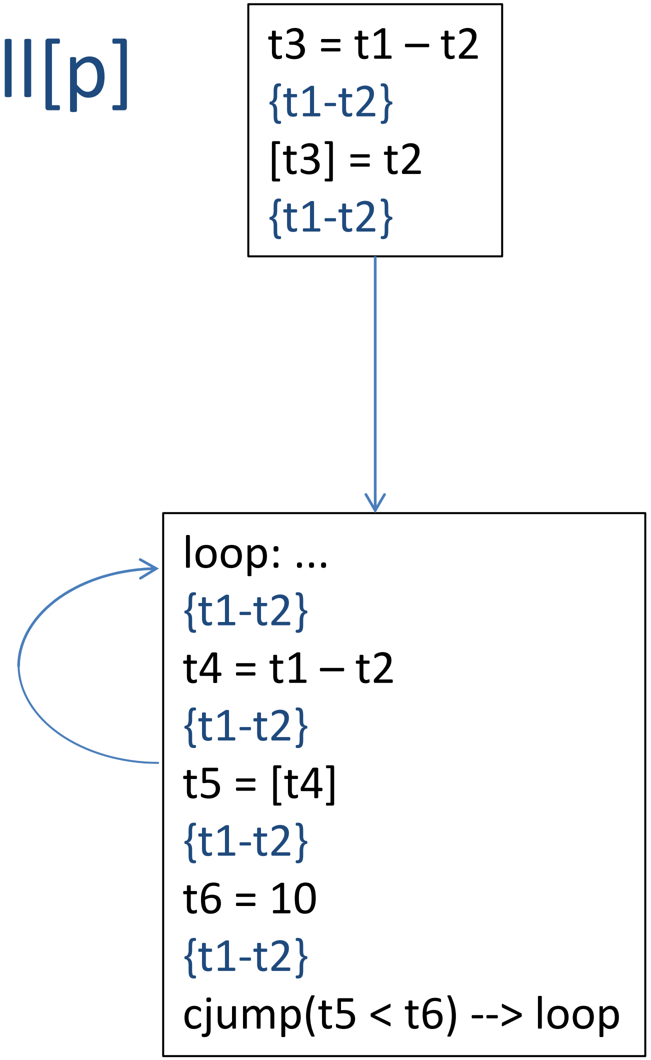
loop: ...
{t1-t2}
t4 = t1 – t2
{t1-t2}
t5 = [t4]
{t1-t2}
t6 = 10
{t1-t2}
cjump(t5 < t6) --> loop

Résolution, itération 5

$$Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$$

$$In[p] = \cap_{q \in pred(p)} Out[q]$$

Point fixe atteint



Analyses de flot de données

	May	Must
Backward	Vivacité $In[p] := (Out[p] \setminus kill[p]) \cup gen[p]$ $Out[p] := \cup_{s \in succ(p)} In[s]$	
Forward		Expressions disponibles $In[p] = \cap_{q \in pred(p)} Out[q]$ $Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$

Analyses de flot de données

	May	Must
Backward	Vivacité $In[p] := (Out[p] \setminus kill[p]) \cup gen[p]$ $Out[p] := \cup_{s \in succ(p)} In[s]$	$In[p] := (Out[p] \setminus kill[p]) \cup gen[p]$ $Out[p] := \cap_{s \in succ(p)} In[s]$
Forward	$In[p] = \cup_{q \in pred(p)} Out[q]$ $Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$	Expressions disponibles $In[p] = \cap_{q \in pred(p)} Out[q]$ $Out[p] = (In[p] \cup gen[p]) \setminus kill[p]$

Plan

- Optimisations locales
  - Flot de données local
- Optimisations globales
  - Flot de données global
  - Forme SSA

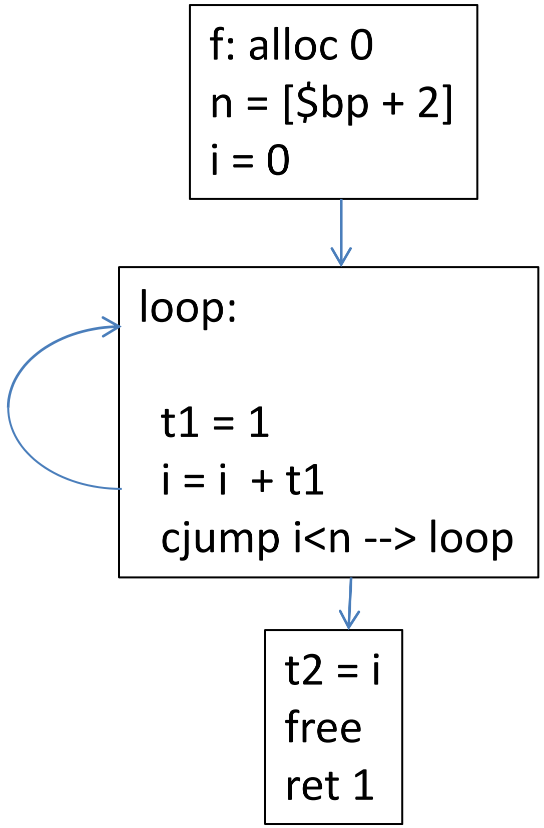
Forme SSA

- Forme normale du CFG avec un flot de données explicite
- La plupart des optimisations sont sous SSA
- Méthode:
  - Passer le CFG sous forme SSA
  - Optimiser
  - Sortir le CFG de la forme SSA

Forme SSA

Static **S**ingle **A**ssignment:

Temporaires définis au plus une fois dans le texte




```
int  f(int n) {
  int i=0;
  do
    i = i + 1;
  while(i<n)
  return i;
}
```

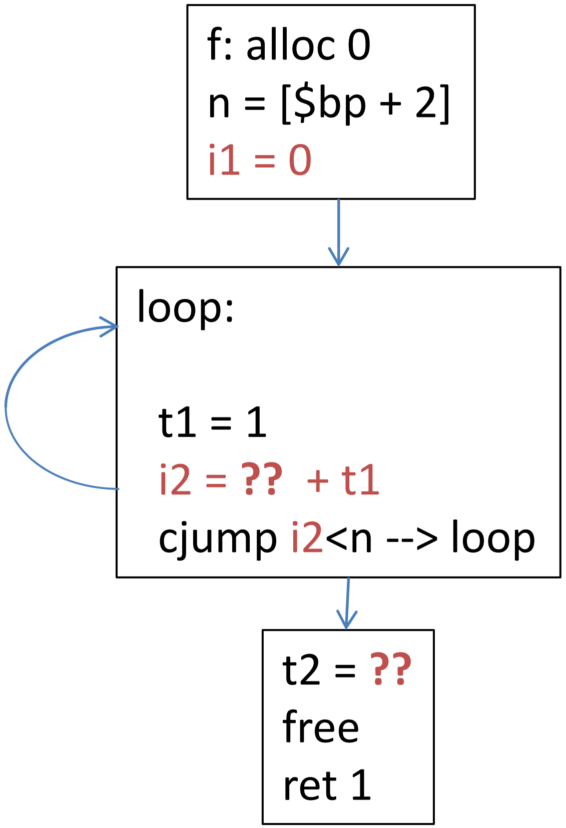
$\rho(n) = n$

$\rho(i) = i$

Forme SSA

Static Single Assignment:

Temporaires définis au plus une fois dans le texte



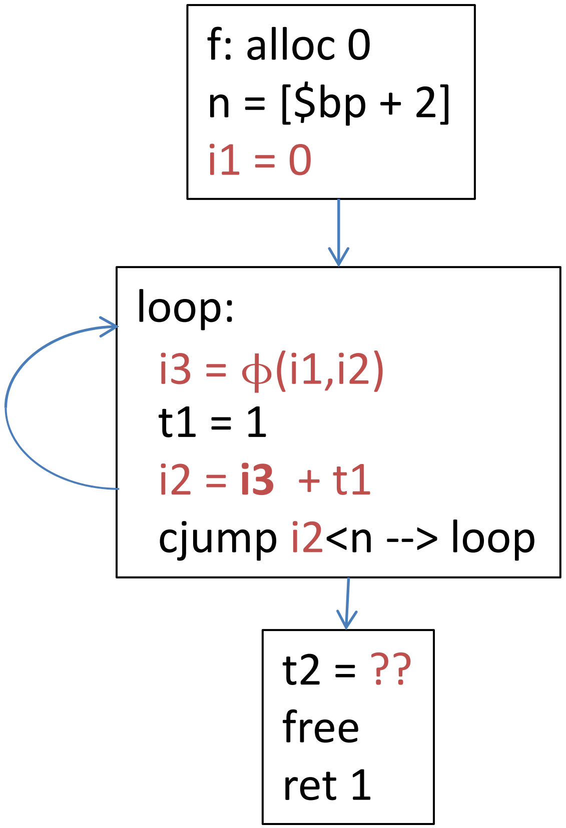

```
int  f(int n) {
  int i=0;
  do
    i = i + 1;
  while(i<n)
  return i;
}
```

ρ(n) = n
ρ(i) = i

Forme SSA

Static Single Assignment:

Temporaires définis au plus une fois dans le texte




```
int  f(int n) {
  int i=0;
  do
    i = i + 1;
  while(i<n)
  return i;
}
```

$\rho(n) = n$

$\rho(i) = i$

Forme SSA

Static **S**ingle **A**ssignment:

Temporaires définis au plus une fois dans le texte

```
f: alloc 0
n = [$bp + 2]
i1 = 0
```

```
loop:
  i3 = ϕ(i1,i2)
  t1 = 1
  i2 = i3  + t1
  cjump i2<n --> loop
```

```
t2 = ??
free
ret 1
```

```
int  f(int n) {
  int i=0;
  do
    i = i + 1;
  while(i<n)
  return i;
}
```

$\rho(n) = n$
$\rho(i) = i$

A-t-on besoin d'une ϕ-fonction?

Forme SSA

Static Single Assignment:

Temporaires définis au plus une fois dans le texte

```
f: alloc 0
n = [$bp + 2]
i1 = 0
```

```
loop:
  i3 = φ(i1,i2)
  t1 = 1
  i2 = i3  + t1
  cjump i2<n --> loop
```

```
t2 = i2
free
ret 1
```

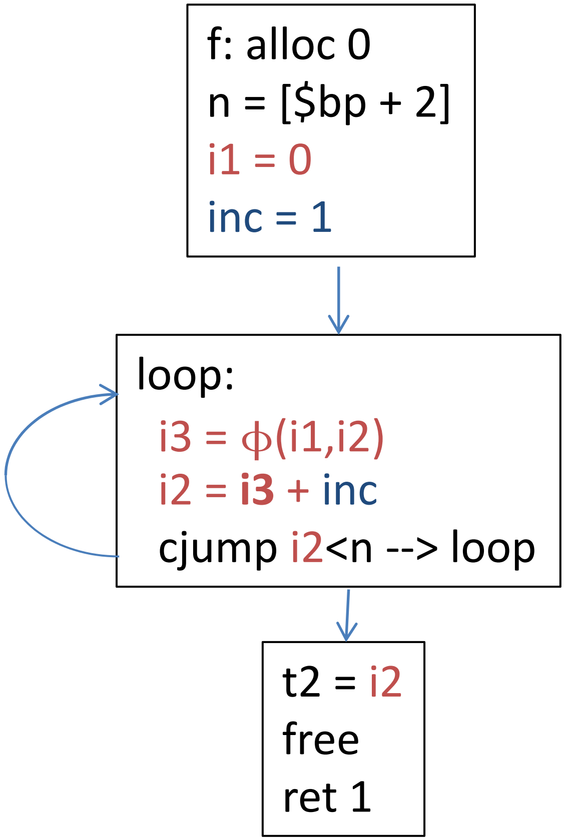
```
int  f(int n) {
  int i=0;
  do
    i = i + 1;
  while(i<n)
  return i;
}
```

$\rho(n) = n$

$\rho(i) = i$

Applications

- Propagation de constantes




```
int  f(int n) {
  int i=0;
  int inc = 1;
  do
    i = i + inc;
  while(i<n)
  return i;
}
```

$\rho(n) = n$

$\rho(i) = i$

$\rho(inc) = inc$

Applications

- Propagation de constantes

```
f: alloc 0
n = [$bp + 2]
i1 = 0
inc = 1
```

```
loop:

  i3 = φ(i1,i2)
  i2 = i3 + inc
  inc = 1
  cjump i2<n --> loop
```

```
t2 = i2
...
```

```
int  f(int n) {
  int i=0;
  int inc = 1;
  do
    i = i + inc;
    inc = 1;
  while(i<n)
  return i;
}
```

$\rho(n) = n$
$\rho(i) = i$
$\rho(inc) = inc$

Applications

- Propagation de constantes

```
f: alloc 0
n = [$bp + 2]
i1 = 0
inc1 = 1
```

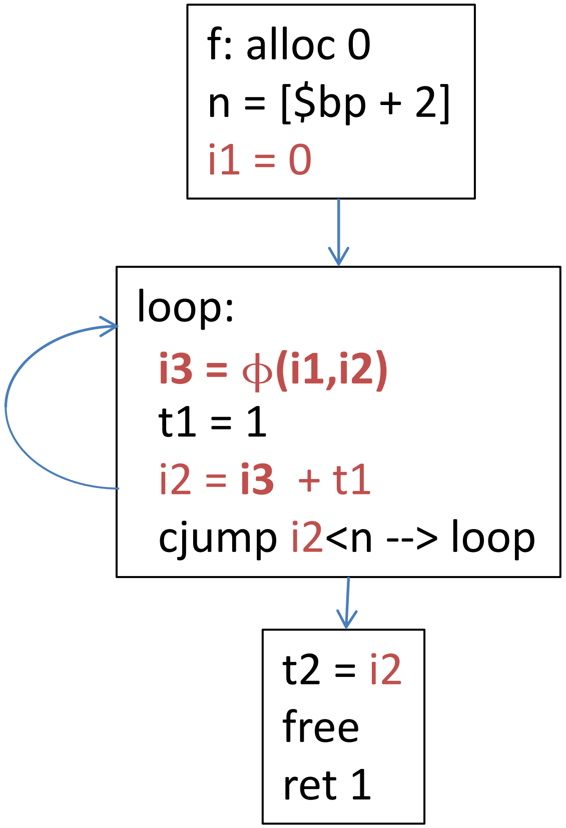
```
loop:
  inc3 = φ(inc1,inc2)
  i3 = φ(i1,i2)
  i2 = i3 + inc3
  inc2 = 1
  cjump i2<n --> loop
```

```
t2 = i2
…
```

```
int  f(int n) {
  int i=0;
  int inc = 1;
  do
    i = i + inc;
    inc = 1;
  while(i<n)
  return i;
}
```

$\rho(n) = n$
$\rho(i) = i$
$\rho(inc) = inc$

Sortie de la SSA

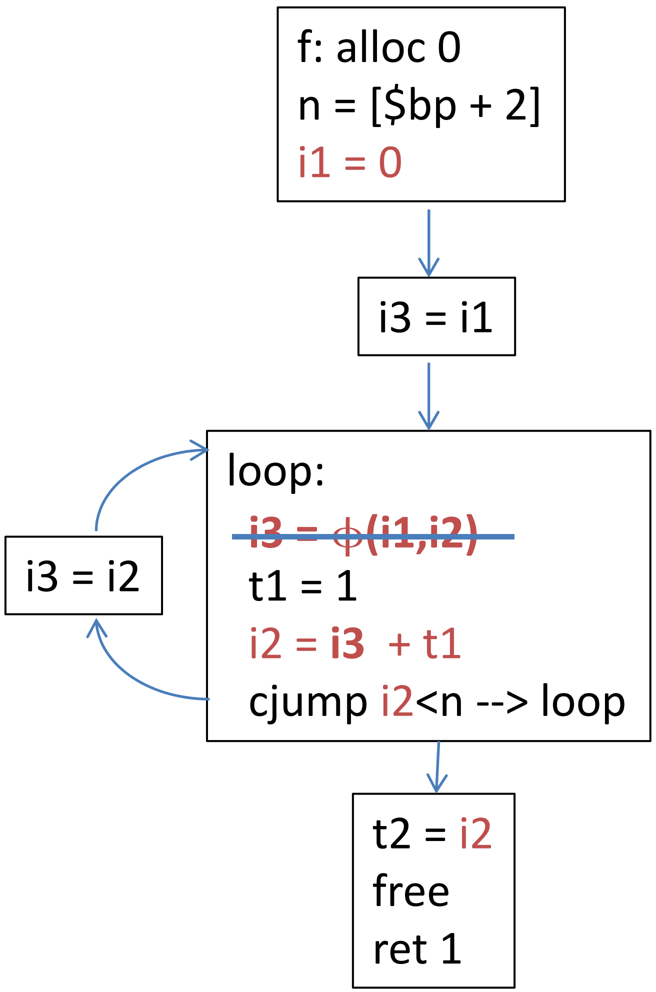



```
int  f(int n) {
  int i=0;
  do
    i = i + 1;
  while(i<n)
  return i;
}
```

$\rho(n) = n$

$\rho(i) = i$

Sortie de la SSA




```
int  f(int n) {
  int i=0;
  do
    i = i + 1;
  while(i<n)
  return i;
}
```

$\rho(n) = n$
$\rho(i) = i$

Et ensuite?

Pour chaque bloc de base:

- On produit le DAG

On soumet au back-end